Modular Formal Islands: Embed theory in your practice

Emilie Balland, Claude Kirchner, Pierre-Etienne Moreau, Anderson Santana de Oliveira

Modular Formal Islands: Embed theory in your practice

Émilie Balland[1], Claude Kirchner[2], Pierre-Etienne Moreau[2], and Anderson Santana de Oliveira[2⋆]

[1] Université Henri Poincaré & LORIA⋆⋆
[2] INRIA & LORIA

Abstract. Motivated by the proliferation and usefulness of Domain Specific Languages as well as the demand in enriching well established languages by high level capabilities like modularity, pattern matching or strategic rewriting, we have introduced in previous works the *Formal Islands* framework.
The main idea consists in integrating, in existing programs, formally defined parts called Islands, on which proofs and tests can be meaningfully developed. Then, *Formal Islands* could be safely dissolved into their hosting language to be transparently integrated in the existing user environment. We present this generic framework and we show that language extensions like Mhtml—providing modular constructions for html— or Tom—a Java language extension allowing for pattern matching and rewriting—are indeed Islands and can therefore be used to embed formal software developments into legacy code.

1 Introduction

Formal methods have shown their importance for the specification, development, and verification of critical systems like real time control of nuclear power plants, embedded systems in airplanes, etc. Surveys show how the huge investments spent in formal methods was returned when they helped identifying bugs and saved enormous resources in terms of humans lives and money [4].

Formal methods are also of main interest to improve the quality of systems used by many other domains of human activity, which are not yet in position to adopt these methods and tools, mainly because this choice demands training personnel, acquiring new and expensive software tools, recoding legacy software, etc. But in practice, such an impact would be greatly justified by the gains formal developments would provide.

Another difficulty in adopting formal methods is that most of the tools provided have limited communication mechanisms with the external world: they are shipped as black boxes, and are hard to integrate with the systems running inside the organization - many function in a kind of batch processing. It is much more interesting to embed formally defined parts of the system in the full operational environment of the computers in an enterprise. That means the formal operation performed by the system is interconnected full time with networks, data bases, computer grids, to mention a few. This increasing need for integration can not be neglected and the problem lies in finding a good balance for combining formal languages and existing ones to create safer systems, since it is clearly not realistic to develop new complete formal systems from scratch to offer all services available nowadays.

In this paper, we present a formalism targeted to the formal development of language extensions. The basic idea is to improve the expressive power of an existing programming language with high level constructs, in a declarative programming paradigm, for example. These constructs only manipulate objects created inside the formal core, but using information from the outer environment where it is lying to built them. We named *Formal Islands* the language extensions with these characteristics - using a analogy where the ocean language (ol) is the

⋆ CAPES BEX-2120/03-8
⋆⋆ UMR 7503, CNRS-INPL-INRIA-Nancy2-UHP

underlying programming language being extended, and island language (il) is the set of formal constructs introduced to it.

In order to reuse the ocean language's compilers, debbugers, and other related tools, the code written in il + ol language (oil) is first transformed into pure ol code. The methodology we propose shows how to obtain a translation process that at the same time does not impose any dependency on the new constructs, and assures the generated structures in ol to preserve the properties of the respective il ones.

As long as this kind of language extensions has a formal semantics, we can state properties about the systems it declares, and the proofs of these properties can be meaningfully developed. Additionally, the notion of *Modular Formal Islands* intends not only to reuse the formal artifacts to build larger systems, but also the proofs associated with them, under the conditions the composition operations preserve the properties of the systems in question.

As instances of this paradigm we can mention the Tom language [11] - which adds pattern matching primitives and strategic rewriting [**?**,**?**,**?**] to existing imperative languages like Java or C. Pattern-matching is directly related to the structure of objects and therefore is a very natural programming language feature, commonly found in functional languages like ML. We also show why Mhtml [7] is an instance of this framework. It has been designed to add modularity constructs to html, enabling the composition of well-formed pages, and resulting of pure, compliant with standards html.

Outline of the paper: we informally present the framework of formal islands by illustrating its main advantages through examples. In the following we state what are the requirements to develop new formal islands adapted to each new necessary language extension. Next, we present how to formalize the phases of the language extension. In the last section we show some examples of actual formal islands developed in the Protheo project.

2 Development Stages

This section presents the development life cycle of programs written using Formal Islands. As shown in Fig. 1, the Island life cycle is composed of 4 phases:

- *anchor* which relates the grammars and the semantics of the two languages,
- *construction* which inserts some Island code in an Ocean program,
- *proofs* or program transformations on islands,
- *dissolution* of the islands in the Ocean language.

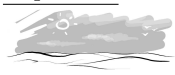
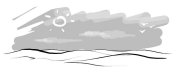
In pictures					
Existing code	Anchor	Construction	Proofs	Dissolution
For example				
Java	ADT	Rewrite rules	Termination	Compilation

Fig. 1. Formal Islands in picture

The anchoring step consists in defining the grammar and semantics of the Island language and in relating it to the existing Ocean one. This step should in particular take care of the data representation correspondence between the Island and Ocean constructions. For instance, when defining a new abstract data type on the Island its correspondence in the Ocean should be made explicit.

The construction phase consists in writing programs in the combined Island and Ocean languages. For example, in Tom, it is possible to define functions using matching constructs (of the form `%match pattern -> JavaCode` or to use term rewrite rules (of the form `%rule term -> term`.

During the proof phase a user can verify that formal properties hold for the island programs. For example, by defining in Tom a set of rewrite rules on top of Java, one could check at that step the termination of the rewrite system, therefore ensuring a better confidence in the program behavior. Modular formal islands allow to compose island programs and to reuse their associated proofs, by identifying the conditions where the properties of the programs are preserved. Again, in the case of term rewrite systems in Tom, there are well know positive results for the modularity of termination under some syntactical restrictions[13,10,5].

The last phase is dissolution. This means that the framework should provide a compilation of Island built programs (that may embed Ocean subparts) into pure Ocean ones. For example again, in Tom, a set of rewrite rules will be compiled into a Java program implementing the normalization process for these rules. Of course the framework setting should ensure that the properties proved at the Island level are still valid after dissolution for the concerned Ocean code.

3 Advantages of Formal Islands

In this section we informally present the main advantages of using the formal islands mechanisms in order to improve the overall quality of programs written in the combination of formal and programming languages: expressivity, modularity, development of proofs, and evolvability.

Improved expressivity The main interest in the design of new language extension or domain specific language (DSL) is to better express data structures and algorithms than general purpose programming languages. Well-designed extensions allow data descriptions and algorithms to become much more concise and understandable for a certain application domain. Some kinds of language extensions concern the transfer of knowledge from the formal methods community to the industry. This scenario can profit the most from the concept of formal islands, because the mathematical background behind the constructs being introduced to an existing language, clearly fits our framework. To cite some examples of high level (formal) constructs embedded in programming languages, from different independent initiatives, let us cite Polymer - an extension of Java with security policies [3], Tom [12]- that introduces pattern matching and more to Java, C, and ML, and Mhtml [7] - a combination of (X)HTML and a set of special tags indicating how to compose web pages, besides, many other formal language extensions can be envisioned.

Formal Proofs Facilitating the statement of proofs of properties is one of the most important benefits formal methods can provide. In a Formal Islands setting, proofs can be realized in two different levels. The first is to build proofs over the terms of the il constructs since its abstractions allow to simplify most of the mathematical foundations necessary to express proofs.

The second is in the process of validation of the anchoring and dissolution - it is important to certify the compilation process to show the resulting pure ol program from dissolution really implements what it is supposed to do, for example, in [8] the authors formalize the behavior of compiled code and define the correctness of compiled code with respect to pattern-matching behavior. These ideas may contribute to implement the necessary infrastructure to proof-carrying code in a near future.

Modular development Modularity is a key feature at design, programming, proving, testing, and maintenance, as well as a must for reusability. The Formal Islands framework can be used to

accomplish modularity among the islands components, but also to improve the modularity of the ol programs. In this sense, it is necessary to study carefully the semantics of composition operators for the il artifacts, and how they can be related with the existing structuring mechanisms from ol. Another concern that has to be addressed to obtain Modular Formal Islands is whether the proofs associated to modules can be reused, in other terms, whether the properties associated to a program are preserved by composition. Mhtml is an approach to put these aspects together, hence modules are checked for conformity with respect to the web standards, in a lightweight implementation of Formal Islands.

Independence One of the appealing feature of Formal Islands is that one can profit from the best of formal approaches without having to throw away the hundreds of millions of LOC running today. Instead, it makes possible to evolve legacy applications in a controlled way. Additionally, after dissolution, the users of oil do not depend anymore on this framework, because the produced code is compatible with the existing environment it always run.

4 Characterization of Formal Islands

The two critical phases of Formal Islands are the anchor and dissolution. Thus, anchor makes the connection between the il and ol languages and dissolution describes the translation from oil to complete ol programs.

4.1 Anchoring

Given two languages il and ol, we introduce the notions of *syntactic anchor* and *representation mapping*, which make a connection between il and ol syntactically and semantically. First, it is necessary to introduce the *syntactic anchor* which links the two grammars to allow us to include islands in ocean programs. Thus, we obtain a grammar for the oil language. The second step of anchorage is to map each object from the Island language with a representation in the set of Ocean objects $\mathcal{O}_{\mathsf{ol}}$ and in this way, to give a semantics to the oil language from the semantics of the ol and il languages.

Syntax il and ol languages are characterized by a grammar (respectively $\mathcal{G}_{\mathsf{ol}}$ and $\mathcal{G}_{\mathsf{il}}$). A grammar is a tuple $\mathcal{G} = (A, N, T, R)$ where A denotes the axiom, N and T, disjoint finite sets of respectively, non-terminal and terminal symbols, and R a finite set of production rules of the form $N \rightarrow (N \cup T)^*$. Given two grammars the *syntactic anchor* is a function `anch` that associates ol non-terminals to il non-terminals, to obtain ol programs with il parts. There may exist two types of anchors corresponding to two types of islands. One called *simple island*, corresponds to pure il constructs and the other called *islands with lakes*, corresponding to islands which can recursively contain ol constructs. In some cases, it is interesting to allow the embedding of Ocean constructs inside Island code. We call *lakes* such constructs that are not modified by the dissolution phase. In term of syntactic anchor, this means that the il grammar can use non-terminals from $\mathcal{G}_{\mathsf{ol}}$.

Example 1. As a first example, let us consider the two grammars, $\mathcal{G}_{\mathsf{ol}} = (\{A\}, \{A\}, \{a\}, \{(A ::= a), (A ::= AA)\})$ and $\mathcal{G}_{\mathsf{il}} = (\{B\}, \{B\}, \{b\}, \{(B ::= b)\})$. The language $\mathcal{L}(\mathcal{G}_{\mathsf{ol}})$ is the set of sequences `a`, `aa`, `aaa`, ... The language $\mathcal{L}(\mathcal{G}_{\mathsf{il}})$ contains only `b`. By considering the *simple syntactic anchor* $\texttt{anch}(\mathcal{G}_{\mathsf{ol}}, \mathcal{G}_{\mathsf{il}}) = \{(A ::= B)\}$ we define the language $\mathcal{L}(\mathcal{G}_{\mathsf{oil}})$ which consists of words like `a`, `b`, `aa`, `bb`, `ab`, and more generally of any sequence of `a` or `b`.

Semantics As for the syntax, we assume given a semantics definition for each language. In the most general case, the objects manipulated by these two languages are not of the same nature. For example, the Ocean language can manipulate tuples and the Island language, algebraic terms. Before giving a semantics to the extended language, we have to make precise the data-structure representations of Island objects in Ocean (the *representation mapping*) and how the data-structure properties in il are mapped to data-structure properties in ol. Given a set of Island objects $\mathcal{O}_{\mathsf{il}}$ and a set of Ocean objects $\mathcal{O}_{\mathsf{ol}}$, a *representation mapping* $\lceil\ \rceil$ is an injective mapping from $\mathcal{O}_{\mathsf{il}}$ to $\mathcal{O}_{\mathsf{ol}}$.

Example 2. Consider the implementation of XML, where to each document corresponds a DOM (Document Object Model) representation. The Java API for DOM defines two main classes for manipulating a document: `Node` and `NodeList`. We could define a mapping from DOM types to abstract algebraic sorts: `Element` and `ElementList`. Thus, a `Node` object becomes a ternary operator `Element` whose first subterm is the name of the XML node, the second subterm is a list of attributes and the third subterm is a list of subterms (which correspond to XML sub-elements). The second and the third elements are terms of sort `ElementList` (because they are implemented by `NodeList` objects in DOM).

Thus, when considering the `<A></A>` XML document, the corresponding algebraic term is `Element("A",[],[])`, where `[]` denotes the empty list. Similarly,

```
<A><B attribute="name"></B></A>
```

is encoded into `Element("A",[],[Element("B",[Attribute("attribute","name")],[])])`. Then, the corresponding mapping is $\lceil \mathit{Element} \rceil$ = `org.w3c.dom.Node` and $\lceil \mathit{ElementList} \rceil$ = `org.w3c.dom.NodeList`. This is indeed one of the representations used by Tom to manipulate and transform XML documents.

The notion of representation mapping establishes a correspondence between data structures in the Island and their representation in the Ocean language. However, we did not put any constraint on the representation of objects. In particular, the function $\lceil\ \rceil$ does not necessarily preserve structural properties of Island objects. In practice, it is the responsibility of the language extension designer to create mechanisms to ensure that the representation mappings actually fulfill this requirement. This can be obtained via a series of predicates equivalently defined over both $\mathcal{O}_{\mathsf{il}}$ and $\mathcal{O}_{\mathsf{ol}}$ objects to test their expected structure correspond to each other.

4.2 Dissolution

Instead of building a new compiler from scratch, we consider a *dissolution* phase which replaces Islands constructs by Ocean constructs. With such an approach, an existing Ocean compiler could be reused. This induces in particular that the use of the Island formalism does not induce a dependence of the user on the island language and tools: after dissolution, the user is again in its original ocean language and can take the benefit of the generated code without depending on run-time libraries or Island language update and maintenance.

At the syntax level, the dissolution step consists of replacing all the il constructs that appear in the ol AST by ol constructs, in order to obtain a complete ol AST. Given two grammars $\mathcal{G}_{\mathsf{il}}$ and $\mathcal{G}_{\mathsf{ol}}$, we call *dissolution* a function $\mathtt{diss} : \mathsf{AST}(\mathcal{G}_{\mathsf{il}}) \rightarrow \mathsf{AST}(\mathcal{G}_{\mathsf{ol}})$ where $\mathsf{AST}(\mathcal{G})$ correspond to the set of abstract syntactic trees of a grammar $\mathcal{G}$.

As the representation mapping has to preserve structural properties, the dissolution (like any compilation phase) must preserve the semantics of the il language. Given il and ol semantics, the ol constructs that are generated must have the same evaluation as the il constructs that they replace.

All these constraints give a first idea of properties that the Island should fulfill to be *Formal* and to preserve proofs. The complete definition of Formal Islands is given in [2].

5 Examples of Formal Islands

We now briefly present Tom and Mhtml, two language extensions developed and maintained in the Protheo Project, that are instances of the Modular Formal Islands framework.

Although there are a number of independent approaches that also fit the definition for Formal Islands. They generally correspond to Domain Specific Languages (DSL). In [14], Spinellis propose eight recurring patterns to classify DSL design and implementation. One of this pattern is the *piggyback pattern* which corresponds to the design of Island languages. The piggyback structural pattern uses the capabilities of an existing language as a hosting base for a new DSL.

An example of DSL implemented by piggyback is SQLJ [9] which is Structured Query Language (SQL) embedded in the programming language Java where the representation mapping between SQL objects and Java objects is given by conversions from SQL types to Java types. The advantages of using SQLJ instead of pure Java are that its translator provides type-checking and schema-object-checking to detect syntax errors and missing or misspelled object errors in SQL statements at translation time rather than at runtime. Programs written in SQLJ are, therefore, more robust than JDBC programs.

Another example we can mention is Linj [1] which is a new language in the Lisp tradition and was invented to allow Lisp programmers to quickly develop and extend Java programs - Linj has object features, and the data-structures between the two languages are shared, making the representation mapping trivial because all Java types are recognized as Linj types.

5.1 Tom

Tom [3] is a *formal island framework* which adds pattern-matching facilities to imperative languages such as C and Java. As presented in [12], a Tom program is a program written in a host language and extended by several new constructs which offer syntactic matching, associative matching with neutral element, conditional rewriting, support for built-in data-types, XML transformation facilities, and a modular strategy library *à la* Stratego [16] which allows to define complex recursive normalizing and traversal strategies. Tom has been used to implement various applications from deep inference in the calculus of structures[6] to the generation of web servers using XML for example. One of the biggest application is the Tom compiler itself, written in Tom and Java.

It is not the purpose of this paper to present the language in detail. In the following, it is sufficient to consider that Tom provides three main constructs:

- `%op` allows to define an algebraic signature (i.e. names of constructors with their profile),
- `%match` corresponds to an extension of `switch/case`, well known in functional programming languages,
- ‘ (backquote construct) allows to build an algebraic term from the host language.

Therefore, a program is a list of Tom constructs (the Islands) interleaved with some sequences of characters (the Ocean). During the compilation process, all Tom constructs are dissolved and replaced by instructions of the host-language, as it is usually done by a preprocessor. From this point, we consider that the Ocean language is Java and we call JTom this specialized version of Tom.

[3] `http://tom.loria.fr`

The following example shows how a simple symbolic computation (addition) over Peano integers can be defined. This supposes the existence of a data-structure and a mapping (defined using %op) where Peano integers are represented by *zero* and *successor*: the integer 3 is denoted by *suc(suc(suc(zero)))* for example.

```
public class PeanoExample {
  %op Term zero() { ... }
  %op Term suc(Term) { ... }
  ...
  Term plus(Term t1, Term t2) {
    %match(t1, t2) {
      x,zero   -> { return `x; }
      x,suc(y) -> { return `suc(plus(x,y)); }
    }
  }
  void run() {
    System.out.println("plus(1,2) = " + plus(`suc(zero),`suc(suc(zero))));
  }
}
```

In this example, given two terms t_1 and t_2 (that represent Peano integers), the evaluation of `plus` returns the sum of t_1 and t_2. This is implemented by pattern matching: t_1 is matched by x, t_2 is possibly matched by the two patterns *zero* and $suc(y)$. When *zero* matches t_2, the result of the addition is x (with $x = t_1$, instantiated by matching). When $suc(y)$ matches t_2, this means that t_2 is rooted by a *suc* symbol: the subterm y is added to x and the successor of this number is returned, using the ` construct. The definition of `plus` is given in a functional programming style, but the `plus` function can be used in Java to perform computations. This example illustrates how the %match construct can be used in conjunction with the considered native language. We can notice that JTom programs contain lakes (the right part of a rule is a Java statement). Note also that lakes can contains Islands, introduced by ` for example.

To give the flavor of the dissolution function for JTom, which corresponds to the compilation phase, we just provide the dissolution of the `PeanoExample` program given previously.

```
  public Term plus(Term t1, Term t2) {
    Term tom_x = t1;
    if (is_fsym_zero(t2)) {
      return tom_x;
    } else if (is_fsym_suc(t2)) {
      Term tom_y = subterm_suc(t2,1);
      return make_suc(plus(tom_x,tom_y));
    }
  }
```

5.2 Mhtml

A first attempt to characterize modular formal islands is Mhtml [7], which is a combination of the regular HTML markup with structuring tags that allows the composition of web documents, through the reuse of code fragments. Mhtml provides the notion of module, importation and template. Modules can be required to be well-formed according to the W3C standards when they are joined together, in order to produce compliant (X)HTML pages. For instance, the code presented in Fig. 2 was extracted from its website[4], and corresponds to the template used to generate all pages found on-line.

[4] `http://mhtml.loria.fr`

```
<?xml version="1.0" encoding="UTF-8"?>
<module name="template">
<params>
<param mode="valid"> pheader </param>
<param mode="valid"> pfooter </param>
<param mode="valid"> pmenu </param>
<param mode="valid"> pcontent </param>
</params>

<html xmlns="http://www.w3.org/1999/xhtml" lang="eng" xml:lang="en">
<head>
<title> Modular HTML </title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<link href="style.css" rel="stylesheet" type="text/css" />
</head>
<body>
   <div class="sheader"> <use> pheader </use> </div>
  <div class="smenu">    <use> pmenu </use> </div>
   <div class="scontent"> <use> pcontent </use></div>
  <div class="sfooter">   <use> pfooter </use></div>
</body> </html> </module>
```

Fig. 2. Exemples of a Mhtml template

The dissolution phase of Mhtml performs text expansions recursively over the oil code, and replaces all occurrences of the modularity constructs by actual parameters, which are indeed concrete module names. For example, Fig. 3 illustrates the result of applying the template presented above to generate the documentation page for the tool's website. The Mhtml compiler checks for circularity, avoiding the process to enter infinite loops, and it was written in JTom.

6 Conclusion

We have discussed in this overview paper the notion of Formal Islands to provide a general framework allowing language designers to base formal language extensions. The main advantage of this approach is to introduce formal methods developed progressively into the applied areas of computer science.

Of course such a framework should be closely linked to proving tools adapted to the properties to be checked: another direction that we are also investigating. For *modular Formal Islands* there is ongoing work towards integrating the positive results on the preservation of modular properties of term rewriting systems, like the work pioneered by Toyama [15] and followed by many others [5,10,13] into Tom, what will be a significant improvement of the framework.

Acknowledgements: Many thanks to the Protheo team and most particularly to Hélène Kirchner and Antoine Reilles for their constructive interactions on this topics.

References

1. João Cachopo António Menezes Leitão. re-ProCLessing: Embedding Lisp within Java. In *International Lisp Conference*, 2003.
2. Emilie Balland, Claude Kirchner, and Pierre-Etienne Moreau. Formal islands, 2006. submitted for AMAST, available in `http://hal.inria.fr/inria-00001146`.
3. Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with polymer. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 305–314, New York, NY, USA, 2005. ACM Press.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="eng" xml:lang="en" xmlns="http://www.w3.org/1999/xhtml">
<head>
<title> Modular HTML </title>
<meta content="text/html; charset=UTF-8" http-equiv="Content-Type"/>
<link href="style.css" rel="stylesheet" type="text/css"/> </head>

<body>

<div class="sheader">
     <h1> MHTML Homepage </h1> </div>

<div class="smenu">
<h2 class="hmenu"> MENU </h2>
<ul>
   <li><a href="index.html">Presentation</a></li>
  <li><a href="documentation.html">Documentation</a></li>
  <li><a href="downloads.html">Downloads</a></li>
  <li><a href="links.html">Links</a></li>
</ul> </div>

<div class="scontent">
The following documentation is provided with Modular HTML:
<ul>
<li> <a href="installation.pdf"><strong>The installation notes
        </strong></a> address installation related concerns. </li>
 <li> <a href="user_guide.pdf"><strong>The user's guide </strong></a>
         details how to use the MHTML program.</li>
<li> <a href="language_reference.pdf"><strong>The language reference</strong></a>
       is a complete description of the MHTML language.</li>
</ul> </div>

<div class="sfooter">
   <small>This site is maintained by
  <a href="http://www.loria.fr/~santana">Anderson S. de Oliveira</a></small>
  Last update: 17/01/2006. All rights reserved. <a href="http://www.loria.fr">
 LORIA</a> <a href="http://www.inria.fr">INRIA</a> </div>
</body> </html>
```

Fig. 3. Result of dissolution of Mhtml

4. Edmund M. Clarke and Jeannette M. Win. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
5. Bernhard Gramlich. *Termination and Confluence Properties of Structured Rewrite Systems*. PhD thesis, Universitat Kaiserslautern, 1996.
6. Ozan Kahramanoğulları, Pierre-Etienne Moreau, and Antoine Reilles. Implementing deep inference in TOM. In Paola Bruscoli, François Lamarche, and Charles Stewart, editors, *Structures and Deduction*, pages 158–172, Lisbon, Portugal, July 2005. Technische Universität Dresden. ISSN 1430-211X.
7. Claude Kirchner, Hélène Kirchner, and Anderson Santana. Anchoring modularity in html. In María Alpuente, Santiago Escobar, and Moreno Falaschi, editors, *WWV*, volume DSIC-II/03/05, pages 139–151. Departamento de Sistemas Informaticos y Computacion, Universidad Politecnica de Valencia, 2005.
8. Claude Kirchner, Pierre-Etienne Moreau, and Antoine Reilles. Formal validation of pattern matching code. In Pedro Barahona and Amy P. Felty, editors, *PPDP*, pages 187–197. ACM, 2005.
9. Jim Melton and Andrew Eisenberg. *Understanding SQL and Java Together: A Guide to SQLJ, JDBC, and Related Technologies*. Morgan-Kaufmann, 2000.
10. A. Middeldorp. A sufficient condition for the termination of the direct sum of term rewriting systems. In *Proceedings 4th IEEE Symposium on Logic in Computer Science, Pacific Grove*, pages 396–401, 1989.

11. Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, 2003.
12. Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction, Warsaw (Poland)*, volume 2622 of *LNCS*, pages 61–76. Springer-Verlag, 2003.
13. M. Rusinowitch. On termination of the direct sum of term rewriting systems. *Information Processing Letters*, 26(2):65–70, 1987.
14. Diomidis Spinellis. Notable design patterns for domain-specific languages. *Journal of Systems and Software*, 56(1):91–99, 2001.
15. Y. Toyama. On the church-rosser property for the direct sum of term rewritig systens. *Journal of the ACM*, 34(1):128–143, January 1987.
16. Eelco Visser, Zine el Abidine Benaissa, and Andrew Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming*, pages 13–26, 1998.